ELSEVIER

Information Processing Letters 81 (2002) 169-174

Information Processing Letters

www.elsevier.com/locate/ipl

Comparison of two code scalability tests

Gordon Lyon

Distributed Systems Technologies Group, National Institute of Standards and Technology,
100 Bureau Drive Stop 8951, Gaithersburg, MD 20899-8951, USA

Received 16 May 1996; received in revised form 6 January 1999
Communicated by F.B. Schneider

Keywords: Analysis; Empiricism; Concurrency; Performance evaluation; Scalability; Software

1. Introduction

"All models are wrong, but some are useful."
George Box

When a computer system is expensive to use or is not often available, one may want to tune software for it via analytical models that run on more common, less costly machines. In contrast, if the host system is readily available, the attraction of analytical models is far less. One instead employs the actual system, testing and tuning its software empirically. Two examples of code scalability testing illustrate how these approaches differ in objectives and costs, and, how they complement each other in usefulness.

Concurrent computing requires scalable code [1,8, 12]. Successes of a parallel application often fuel demands that it handle an expanded range. It should do this without undue waste of additional system resources. Definitions of scalability will vary according to circumstances — when looking for speedup, problem size is fixed and the host system grows. In another case, one evaluates an enlarged problem together with a larger host [3]. The discussion that follows assumes no particular scalability metric. As others have commented, "We report our research results only in terms of execution times, leaving the choice of a scalability metric to the user" [10].

SLALOM — the *Scalable, Language-independent, Ames Laboratory, One-minute Measurement* — is a code used here as a concrete example. SLALOM ranks computer systems by the accuracy they achieve on a realistic image rendering problem in radiosity [4]. Accuracy is denned as geometry "patches" computed during a test, which SLALOM adjusts automatically to one minute of execution. By fixing time, SLALOM accommodates a very broad spectrum of host systems. SLALOM'S original patch generation — used here — is $0(N^3)$, a non-linearity that makes interpreting distances between distinct "patch" ratings less intuitive. An *0(NlogN)* patch generation improves comparisons between systems, however, this variant is not so easily ported to new systems. A sequel benchmark, HINT, is "linear in answer quality, memory usage and number of operations" [5].

2. An analytical scalability model

Code scalability can be evaluated analytically via structural models or empirically through multi-dimen-

E-mail address: lyon@nist.gov (G. Lyon).

0020-0190/02/$ - see front matter Published by Elsevier Science B.V.
PII: S0020-0190(01)00204-6

Table 1
Data from SLALOM benchmark

Independent settings		Measured dependent response R (in 'patches')
X	S	
-1	-1	650 (a)
+1	-1	401 (b)
-1	+1	1167 (c)
+1 .	+1	85.3 (d)

X = -1, +1 denote *regular* and *slower* versions, respectively, of the Setup phase of SLALOM.
S = -1, +1 denote 4 and 32 processors, respectively, on an iPSC/860.

sional curve fittings. Science has a long history of results predicted from analytical models constructed upon a prior body of knowledge. The analytical paradigm is:

measure **details** => *model* => *predict* **response**

With software, detailed measurement traces serve as input to a tailored model, which then yields a response corresponding within some accuracy to actual observation. Imagine that a parallel version of SLALOM on a 4-processor iPSC/860 host has been modeled. Checking the model's predicted response of 720 "patches" against an actual performance of 650 (see entry (a) of Table 1) indicates a 10% overestimation of performance. Such checks begin to establish an appropriate level of confidence in the analytic model.

Modeling Kernel, MK, is designed to avoid testing actual codes on critical and expensive production systems. Semi-automated, the tool kit *MK* supports scalability investigations of message-passing applications [10]. The described version allows only deterministic message passing. Such a limitation in generality is not unusual, since sound analytical models may consume significant effort — prudent restrictions diminish modeling costs. Within their chosen domain of deterministic message passing, *MK* models are accurate enough to eliminate any need for full-scale tracing and tuning. The method claims 8-20% error, which is more than adequate for most code development.

2.*1. Use of the toolkit MK*

MK's analysis assumes that inter-processor communication dominates scalability concerns within the chosen class of scientific applications and host architecture. A modeled program becomes a flow structure that, with some delays for computation, principally triggers the sending and receiving of messages. To do this, *MK* generates, decorates and evaluates a structurally condensed program parse tree:

(a) Extract a parse skeleton involving control flow from the program's code and other invoked codes. The skeleton records structure for loops, conditionals, invocations and communications.

(b) Add symbolic expressions and actual known values on various bounds, such as iterations, branching frequencies, message lengths and basic block execution times. Many instances have algebraic loop bounds. Branching frequencies come from actual execution traces. The modeler supplies supplementary data as needed. Data-dependent execution behavior may require a hybrid technique (discussed later).

(c) Augment tree nodes further by attaching communication phase graphs. Each graph relates a number of communication operations that together transport information among a set of processors during a phase. A data exchange exemplifies a synchronous phase. The pipeline distribution of data, on the other hand, typifies an asynchronous phase. An analysis of actual communication traces provides the basis for building the phase graphs.

(d) Evaluate the decorated tree. *MK* can make two distinct evaluations here. The first is a symbolic interpretation of the tree; this yields an algebraic expression for runtime. The second evaluation generates a simulation time trace.

Symbolic evaluation is preferable, since algebraic estimates are cheaper than simulations. However, message-passing depends upon sometimes complex interactions of topology, protocols, dependencies, traffic volume and distribution. Limiting these details in the model saves cost, but carrying this too far risks a complete loss of reliability. For example, with some programs, having slower code for parallel routine A will accelerate completion. This occurs by diminishing the frequency with which the various A's send messages over congested interconnections. Message retransmissions are then fewer. Execution is faster overall. This behavior must be captured in a model for such classes of program and host. It precludes any simple solution that ignores contention. The authors of *MK* are well aware of this, yet it appears that for some of their initial work they can employ the less taxing symbolic evaluation. They remark that introducing simulation to handle more complex modeling causes a pronounced jump in *MK's* processing demands. However, traces from *MK* simulations provide excellent insights into performance shortcomings.

3. An empirical scalability test

Empirical modeling goes from an observable response to specific details:

measure **response** => *model* => *correlate with* **details**

In contrast to the analytic method, the response is mot generated by the model — it is measured off the actual system. Values of the response are correlated against details called factors [6], each factor having a limited number of settings (e.g., option Y set "on" or "off", module versions 5A, 5B or 5C) [2,6]. An observed response with software is some readily accessible characteristic, such as run time or average transactions per unit time [2]. SLALOM's response is its "patches" in a one-minute trial. Given that the host system is readily available but perhaps not well understood, empirical scalability testing can be highly attractive. It is especially powerful at *screening* software factors to find those associated with performance problems [7]i.

3.1. A handy test

The following is an easy method for checking code components for scalability. Source code is not needed, only the ability to patch in delays. The actual system must be available for running the code. Table 1 has measurements of a parallel version of SLALOM, where scalability for component *Setup* is in question. *X* and *S* are independent variables. *X* = -1 denotes the usual code for *Setup*. *X* = +1 denotes a slower *Setup* made by attaching to it an artificial delay. This computationally benign delay must be large enough to show up in measurements of response *R*. Critical to the test, a delay lowers *Setup's* efficiency by adding extra clock cycles. Similarly, S = +1 denotes a larger scale host system (32 processors in Table 1) and *S* = -1 indicates a smaller scale host (4 processors). The measured response, *R*, is a dependent variable. An intuitive scalability test, given Table 1, is:

$$\frac{a-b}{a} \geqslant \frac{c-d}{c} \Rightarrow \text{\textit{Setup} code scales (relatively).} \qquad (1)$$

In essence, as scaling up proceeds, *the fractional drop* in performance from added *Setup* cycles should not matter more than it did initially. Inserting values from Table 1 into (1),

$0.383 \geq 0.269 \Rightarrow$ *Setup* scales relative to SLALOM.

3.2. More from the same measurements

Test (1), above, is sensitive to errors in measurements *a*, *b*, *c* and *d*. Each measurement has variation characterized as standard error, *SE*. *SE* arises from system background and interconnection actions, non-deterministic algorithms, coarse clocks and similar factors. Experience shows distributed-memory as more variable than shared-memory. Observe that a highly variable host dictates a larger inserted delay, which if excessive will distort observations.

Repeating each measurement four times and averaging will halve the standard error to $SE' = SE/\surd 4$. However, since *a*, *b*, *c* and *d* sample the same "noise" distribution whose mean is assumed zero, these four measurements can be manipulated to achieve the improved *SE' without* additional runs.

3.3. EST: an improved Empirical Scalability Test

Consider response *R of* Table I as a transfer function *R(X, S)* with parameters *X* and *S* and a bivariate (Maclauren) expansion about zero [8]:

$$\begin{aligned} R(X,S) = {} & R(0,0) + \sum_{y\in\{X,S\}} y\left[\frac{\partial R}{\partial y}\right]_0 \\ & + \frac{1}{2}\sum_{y\in\{X,S\}} y^2\left[\frac{\partial^2 R}{\partial y^2}\right]_0 \\ & + XS\left[\frac{\partial^2 R}{\partial X \partial S}\right]_0 + \cdots + \cdots . \end{aligned} \qquad (2)$$

For identifying scalability bottlenecks, most higher order terms of the expansion can be *assumed* unimportant and thus *set* identically zero. Restricting interest to four major terms yields an approximate expansion:

$$R(X,S) \cong \mu + \beta_X X + \beta_S S + \beta_{X,S} XS. \qquad (3)$$

The right side of Eq. (3) expresses how performance *R* changes as *X* and 5 assume different settings, as in Table 1. Product *XS,* which is +1 only when *X* and *S* share a common setting, indicates an interaction. Linear in unknowns *μ*, $ß_X$, $ß_S$ and $ß_{X,S}$, Eq. (3) is solved with values taken from Table 1. Parameter

$$\mu = R(0,0) = \tfrac{1}{4}(a+b+c+d)$$

averages all responses to estimate a base state response at (X= 0,S=0). Coefficient

$$\beta_X = \tfrac{1}{4}(-a+b-c+d)$$

is the sensitivity of *R* to *Setup's* change in efficiency. Coefficient

$$\beta_S = \tfrac{1}{4}(-a-b+c+d)$$

expresses R's sensitivity as scale *S* changes — the investigator defines what scale means. SLALOM in Table 1 has more processors *and* more problem calculation at higher levels of scaling. Interaction coefficient

$$\beta_{X,S} = \tfrac{1}{4}(a-b-c+d)$$

expresses how factor *S* affects factor *X*. The four terms define an improved test:

$$\frac{\beta_{X,S}}{\beta_X} \leqslant \frac{\beta_S}{\mu} \Rightarrow \textit{Setup} \text{ code scales.} \qquad (4)$$

Test (4) restricts acceptance of the effect of *Setup's* delay under scaling to at most its current relative importance. Inserting μ = 767.75, $ß_X$ = -140.75, $ß_S$ = 242.25 and $ß_{X,S}$ = -16.25 into (4),

0.115 $\leq$ 0.316 => *Setup* scales relative to SLALOM.

Criterion (4) subsumes the earlier test (1). To see this let

$$F_{(a-b)/a} \equiv \frac{\beta_X \Delta X}{\mu}$$

define the base state fractional drop in performance from *Setup's* delay AX analogous to *(a — b)/a* in test (1). Similarly, let

$$F_{(c-d)/c} \equiv \frac{\beta_X \Delta X + \beta_{X,S}\Delta X \Delta S}{\mu + \beta_S \Delta S} = \left[\frac{1+\frac{\beta_{X,S}}{\beta_X}\Delta S}{1+\frac{\beta_S}{\mu}\Delta S}\right]\frac{\beta_X \Delta X}{\mu}$$

denote the fractional drop in performance at scaling increment ? S caused by delay ? X; this corresponds to the earlier (c — *d)/d.* Condition (4) above assures that the square-bracketed fraction is no greater than unity, so as in test (1),

$F_{(a-b)/a} \geq F_{(c-d)/c}$.

4. *MK* and *EST* compared

Table 2 highlights the quite different perspectives of *MK* and *EST.* Although *MK's* implementation is large, complex and relatively expensive, it also supplies copious details. An *MK* model explicitly expresses pivotal structures and behaviors of the specimen program. *EST is* designed as a quick litmus test that locates bottlenecks without explaining them. It assumes a reasonably well-behaved response surface within its range of test. Extrapolating *EST* results outside of this range poses a risk that must be offset by additional structural and behavioral knowledge.

Demands upon the actual system are heavy for both approaches, but differ in origins and technical details. Avoiding full-scale runs is *MK's raison d'etre.* Lacking structural knowledge, *EST* relies upon actual full-scale execution behavior. It thereby avoids abstraction errors found in an analytic model, but pays with the cost of the runs. *MK* relies upon system tracings from smaller runs. These tracings, which generate model input data, can be vexing and costly: Good instrumentation is not something one can expect on all scalable systems [9].

Complex interactions between specimen and host upset both methods. A performance region with a pronounced non-linear scaling response might throw *EST off* track. Closely spaced host sizes will help, but the cost of using *EST* rises [7]. *MK* can always incorporate further complexity into its model and simulation runs, but this becomes more expensive. Once constructed,

Table 2
Contrasting analytical *MK* and empirical *EST*

	MK	*EST*
Model type	bottom-up — analytical construct built espe-	top-down — empirical fit to a general multi-
Input	fine details	factor settings and responses
Fundamental basis	interpret structure of program and system	correlate measured responses with code/host changes (settings)
Focus	targeted to a specific system	general, for any system
Tool size	large and complex by nature	small
Need for full scale runs	none — reason for prediction model is to avoid them :	must have — lacks structural knowledge of system
Data demands	heavy — detailed trace capture and collection	light — response is easily captured
Demands upon actual system	needs several smaller scale runs with trace enabled	has at least two full-scale runs and two runs at smaller scales
Tool processing demands	heavy — abstract model interpretation	light — solve 4 linear equations in (4) unknowns
Explanatory power	high — especially with dynamic behavior trace	low — treats problem as transfer function
Model state space	large and grows with scale	small and fixed
Display	typical visualization challenges	scalable due to fixed state space

MK's model explores a large number of hypothetical variations at significantly reduced cost. *EST*, relying upon full-scale runs to provide structural information implicitly, retains a pay-by-the-test flavor.

4.1. Hybrids

Each approach can gain from the other. If *EST's* full-scale test runs of a code become impractical, the statistical underpinning can be adapted to use actual production runs. Doing this requires an analysis of the code's structure. Special structural knowledge creates other EST-like opportunities to assay crucial program structures, e.g., the barrier test in [11]. On the other hand, unpredictable, data-dependent control bothers the analytical structure *of MK.* In such circumstance^, analysis of a program's structure supplies few hints for interpretation bounds. *MK* solves this by employing statistical regression to build small (empirical) estimator expressions from data traces [10].

5. Conclusions

Analytical testing incurs modeling expenses but yields a rich output. Empirical testing applies quickly and broadly but provides circumscribed results. Neither approach dominates everywhere. System vendors and software engineers may prefer analytical techniques, from which they learn much. Vendors will amortize high modeling costs over many sales of the software. Service bureaus, on the other hand, just want to install and tune a code. In such circumstances, an empirical approach can make practical sense. Hybrids increase the utility of both methods.

Notes. An earlier version of this paper appeared in the Proceedings of TDP'96 — International Conference on Telecommunication, Distribution and Parallelism, June 1996, La Londe Les Maures, France. Support came from NIST Task 40131 and ARPA Task 7066.

References

[1] M.A.M. Al-Abdulkareem, Scalability analysis of large codes using synthetic perturbations and factorial designs, Dissertation for Doctor of Philosophy, University of Oklahoma, Norman, OK, 1997.

[2] W. Freiberger (Ed.), Statistical Computer **Performance** Evaluation, Academic Press, New York, 1972.

[3] J.L. Gustafson, Reevaluating Amdahl's law, Comm. ACM 31 (1988) 532-533.

[4] J.L. Gustafson, D. Rover, S. Elbert, Slalom.c source text file, Ames Laboratory, Ames, IA, 1990.

[5] J.L. Gustafson, Q.O. Snell, HINT: A new way to measure computer performance, available at http://www.scl.ameslab.gov/Publications/HINT/ComputerPerformance.html (Oct. 1997), 17pp.

[6] R. Jain, The Art of Computer Systems Performance Analysis, John Wiley & Sons, New York, 1991.

[7] G.E. Lyon, R. Snelick, R. Kacker, Synthetic-perturbation tuning of MIMD programs, J. Supercomput. 8 (1994) 5-27. (A companion multivariate statistical analysis tool, *S-Check*, automates much of this. See http://www.scheck.nist.gov/scheck for free copies.)

[8] G.E. Lyon, R. Kacker, A. Linz, A scalability test for parallel code, Software — Practice and Experience 25 (1995) 1299-1314.

[9] A. Mink, W. Salamon, Operating principles of the PCI bus MultiKron interface board, Internal Report, NISTIR 5993, Scalable Parallel Systems and Applications Group, National hist. of Standards and Tech., Gaithersburg, MD, 1997.

[10] S.R. Sarukkai, P. Mehra, R.J. Block, Automated scalability analysis of message-passing parallel programs, IEEE **Parallel** Distributed Technology 3 (1995) 21-32.

[11] R. Snelick, J. JaJa, R. Kacker, G. Lyon, Synthetic-perturbation techniques for screening shared memory programs. Software — Practice and Experience 24 (1994) 679-702.

[12] X.-H. Sun, D. Rover, Scalability of parallel algorithm-machine combinations, IEEE Tran. Parallel Distributed Systems 5 (1994)599-613.